Appears in the Journal of Parallel and Distributed Computing, April 2005

Memory-Side Prefetching for Linked Data Structures for Processor-in-Memory Systems

Christopher J. Hughes
Architecture Research Lab
Intel Corporation
2200 Mission College Blvd., SC12-303
Santa Clara, CA 94054
christopher.j.hughes@intel.com
Phone: (408)765-9454 Fax: (408)653-8157

Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave.
Urbana, IL 61801
sadve@cs.uiuc.edu
Phone: (217)333-8461 Fax: (217)333-3501

Abstract

This paper studies a memory-side prefetching technique to hide latency incurred by inherently serial accesses to linked data structures (LDS). A programmable engine sits close to memory and traverses LDS independently from the processor. The engine can run ahead of the processor because of its low latency path to memory, allowing it to initiate data transfers earlier than the processor and pipeline multiple transfers over the network. We evaluate the proposed memory-side prefetching scheme for the Olden benchmarks on a processor-in-memory system. For the six benchmarks where LDS memory stall time is significant, the memory-side scheme reduces execution time by an average of 27% compared to a system without any prefetching. Compared to a state-of-the-art processor-side software prefetching scheme, the memory-side scheme reduces execution time in the range of 20% to 50% for three of the six applications, is about the same for two applications, and is worse by 18% for one application. We conclude that our memory-side scheme is effective, but a combination of the processor- and memory-side prefetching schemes is best and provide a qualitative framework to determine when either scheme should be used.

Keywords: Prefetching, processor-in-memory, linked data structures

1 Introduction

Linked data structures (LDS) are increasing in importance due to the widespread use of object-oriented programming and application domains that involve large dynamic data structures. Hiding the memory latency incurred in traversals of such data structures, however, is notoriously difficult. Such traversals involve a chain of inherently serial, dependent loads – the next address to be accessed is not known until the data from the previous load returns to the processor. To hide this latency, several researchers have proposed novel prefetching techniques initiated at the processor (e.g., [21, 27, 35, 36, 41]) that we refer to as *processor-side prefetching*. In processor-side prefetching, hardware near each processor prefetches data for that processor from all nodes in the system.

We study *memory-side prefetching*, where a prefetch engine close to memory speculates on LDS data that a (local *or remote*) processor may need from that memory. A software command from the requesting processor encodes a summary of the LDS and the expected traversal. The prefetch engine uses the command to independently perform the traversal, requesting memory to send the traversed data to the processor. Although the prefetch engine's traversal is also serialized, its proximity to memory results in faster service than requests initiated at the processor. This potentially allows the prefetch engine to run ahead of the processor, initiating data transfers earlier than the processor and pipelining multiple transfers over the network.

Key parameters for the effectiveness of memory-side prefetching are the proximity of the prefetch engine to memory and support for address translation. Our goal is to explore the fundamental limitations of and tradeoffs between processor- and memory-side prefetching, necessitating best case conditions for each. The best case for memory-side prefetching would integrate the prefetch engine with memory and address translation hardware. There have been significant recent advances in processor-in-memory (PIM) systems, which integrate the processor with memory [12, 15, 20, 22, 24, 37]. For the large class of applications where a single PIM chip does not provide sufficient memory, systems based on multiple PIM chips have been proposed (e.g., IBM's Blue Gene [15] and Execube [22]). Wallach predicts high performance systems of 2009 will be built solely from multiple PIM chips [38]. For these reasons, we chose to use a multiprocessor built solely from multiple PIM chips as the base system for our quantitative evaluations. The prefetch engine is on the PIM, making it close to memory and allowing it use of the processor's address translation hardware. Despite having memory, the processor, and the prefetch engine on the same chip, the fundamental differences between processor-side and memory-side prefetching remain. This be-

comes especially clear when we consider multiprocessor systems, where each PIM's prefetch engine initiates prefetches for all processors in the system. We also discuss alternative configurations (e.g., non-PIM) using a qualitative framework derived from the quantitative evaluations (Section 5.4).

We perform our evaluation for the Olden suite of pointer-intensive benchmarks [5], a suite commonly used in LDS studies [27, 36, 40]. For the six applications where LDS memory stall time is significant, the proposed memory-side scheme reduces execution time by a mean of 27% (range of 0%-62%) compared to an equivalent system without prefetching. Compared to a state-of-the-art processor-side software prefetching scheme based on jump pointers [27, 36], the memory-side scheme reduces execution time in the range of 20%-50% for three of the six applications, is the same for two applications, and is worse by 18% for one application. We conclude that our memory-side prefetching scheme is effective, but a combination of the processor- and memory-side schemes is best and develop a qualitative framework to determine when each scheme should be used.

The only other work on memory-side prefetching of which we are aware is [40]. That work compares a proposed memory-side engine with a similar engine used for processor-side prefetching, and finds the former to be very effective. Instead, we compare memory-side prefetching to jump-pointer prefetching, a state-of-the-art processor-side scheme that can exploit parallelism in the memory system. Therefore, our results differ significantly from [40]. For example, we show the processor-side scheme does better when there is relatively low computation per LDS node for large LDS. In contrast, the previous work found the memory-side scheme to be the best for all the macro-benchmarks studied, while a micro-benchmark study showed that processor-side prefetching would perform better with large amount of computation per LDS node. Our work also differs because our prefetch engine can capture more LDS traversals (the one in [40] is specific to linked-lists). We discuss previous work further in Section 6.

2 Background on Processor-Side LDS Prefetching

The most promising processor-side LDS prefetching techniques are based on jump-pointers [27, 36]. In this work, we primarily consider the software jump-pointer techniques by Roth and Sohi [36] to represent processor-side prefetching, and discuss them below. Other techniques, including cooperative and hardware jump-pointer prefetching and prefetch arrays, are discussed in Section 6.

With jump-pointer prefetching, an LDS node is augmented with *jump-pointers* that point to nodes that will be accessed multiple iterations or recursive calls in the future. When an LDS node

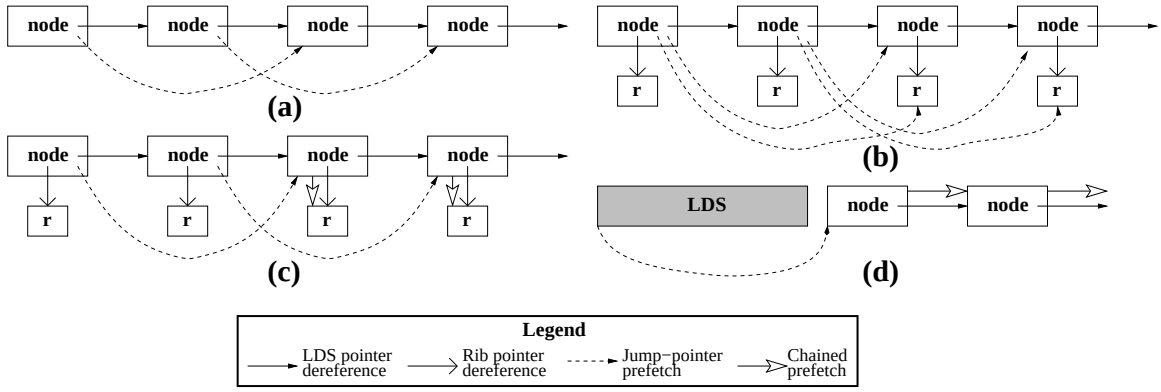


Figure 1: **The four jump-pointer prefetching idioms.** (a) Queue jumping. (b) Full jumping. (c) Chain jumping. (d) Root jumping.

is visited, prefetches are issued for the locations pointed by its jump-pointers. Roth and Sohi propose four different idioms for jump-pointer prefetching [36], as described below.

Queue jumping is the simplest idiom. Each node's jump-pointer points to another node in the same LDS that is likely to be accessed in the near future (Figure 1(a)). The distance between a node and the node pointed by its jump-pointer is called the *jump interval*. As long as this interval is large enough, the jump pointer prefetch will complete before its corresponding demand access. The first few nodes of an LDS, however, have no jump-pointers pointing to them and so are not prefetched. This results in a *startup period* where the processor may incur stall time.

Full jumping is a variant of queue jumping for *backbone-and-rib* structures, which consist of an LDS (backbone) with each node containing one or more pointers to data nodes (ribs). Each node also has jump-pointers to the ribs of another node (Figure 1(b)), allowing ribs to be prefetched in parallel with the backbone.

Chain jumping is a different method of prefetching rib structures that does not incur the overhead of maintaining jump-pointers for ribs. The backbone of the LDS is prefetched using queue jumping, but the ribs are prefetched using built-in (natural) pointers of the original data structure (Figure 1(c)). The rib prefetches are called chained prefetches. Generally, the processor issues chained prefetches from a node in the same iteration that it issues the jump-pointer prefetch for the node. Therefore, if the jump-pointer prefetch does not complete by the end of the iteration, the processor will stall because of the address dependency for the chained prefetch. A solution is to double the jump interval and stagger the jump-pointer and chained prefetches for a particular node and its ribs. We call this *staggered chain jumping*. (This method is not explicitly mentioned

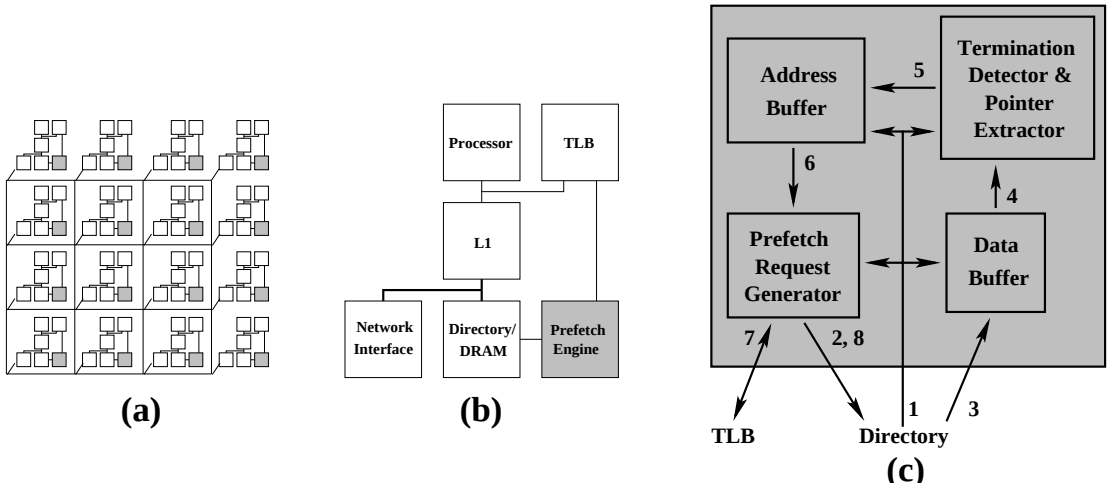


Figure 2: **A PIM with a prefetching engine in a multi-PIM system.** (a) The multi-PIM system. (b) The contents of the PIM. The prefetch engine is highlighted. (c) The contents of the prefetch engine, along with the steps to prefetch an LDS.

in [36], but was used in the prefetch codes we obtained from Roth.)

Root jumping is a variant of chained prefetching. It is applicable to short LDS which are dominated by the startup period and to dynamic LDS where updating jump pointers creates considerable overhead. In root jumping, an entire LDS has a single jump-pointer to the next LDS to be accessed (Figure 1(d)). When an LDS traversal is begun, the jump pointer is used to prefetch the first node of the next LDS. Subsequently, the natural pointers of the second LDS are used to issue prefetches to its nodes in lockstep with the traversal of the first LDS. The use of the natural pointers is like chained prefetching and incurs similar serialization.

3 A Memory-Side LDS Prefetch Engine

3.1 System Architecture

As discussed in Section 1, the underlying base system we consider is a multiprocessor built solely from multiple PIM chips (Figure 2 (a)). Section 5.4 discusses alternative architectures. The PIMs are connected to each other via a conventional multiprocessor network in a directory-based, cache-coherent, release consistent shared-memory organization. Since the focus of this work is not to suggest an optimal hardware organization for a PIM, for simplicity, we use a common model of a multiprocessor node, but place all of the components on the same chip (Figure 2(b)). Other, more

aggressive PIM architectures have been proposed, including [10, 11, 12, 20, 24]. These systems exploit data parallelism to take advantage of the high bandwidth available on PIMs. However, they do little to further decrease memory latencies; therefore, they are unlikely to enable faster LDS traversals than our base architecture.

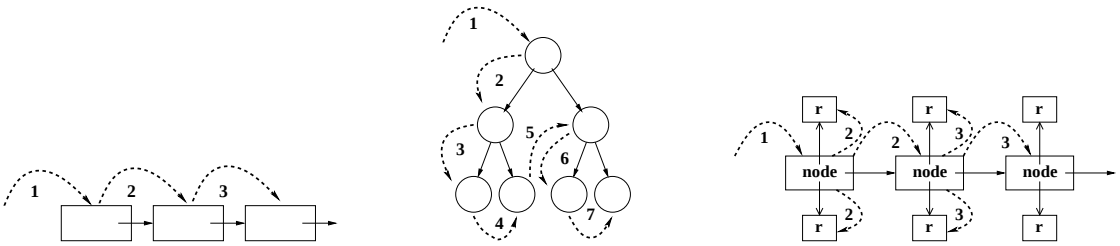
The novel feature of our memory-side prefetching system, and our focus, is a prefetch engine on each PIM chip. The engine sits next to the directory and communicates only through it, but also accesses the processor's TLB (Figure 2(b)). We call this a memory-side engine because it prefetches data from the adjacent memory for all processors in the system. In contrast, a processor-side engine prefetches data from all memories in the system for the adjacent processor. The hardware for the prefetch engine is minor relative to that for a state-of-the-art processor, consisting of only a few buffers, a counter, some comparators, multiplexors, decoders, and some straightforward control logic. The rest of this section provides details on the prefetch engine (Figure 2(c)) and the command to access it.

3.2 Conveying LDS Traversal Information to the Prefetch Engine

The goal of the prefetch engine is to traverse an LDS ahead of the processor and send the data to the processor before its corresponding demand access. This requires information about the LDS structure and traversal path. A general way to convey this information is for the processor to send to the prefetch engine code that can be executed to traverse the LDS, and for the prefetch engine to have the ability to execute such code. In practice, we found that for most of the benchmarks we evaluated, the LDS traversal path depends primarily on the LDS structure and sometimes on the results of simple comparisons involving the data within the LDS. Further, the LDS structure and the comparison operations can often be encoded concisely in a few bytes. Therefore, in this study, we assume special prefetch commands that encode this information, and require the programmer or compiler to insert such commands in the code before an LDS traversal. The result is a much simpler prefetch engine at the cost of some generality. Nevertheless, the traversals captured by this engine are more general than previous work (see Sections 4.2 and 6).

3.3 LDS Types and Traversals Supported

We support three common LDS types – linked lists, trees, and backbone and rib structures. Figure 3 illustrates supported traversals. The dashed arrows and numbers indicate the order of traversal. The other arrows indicate pointers in the data structures. For a list traversal (Figure 3(a)), the



(a)A linked list traversal (b)A DFS tree traversal (BFS & non-deterministic also supported) (c)A "backbone-and-ribs" traversal – the nodes with r's are the ribs

Figure 3: **Supported LDS and traversal types.** The dashed arrows with numbers indicate the order of traversal. The other arrows are pointers in the data structures.

prefetch command simply needs to provide the offset into a node for the "next" pointer. Our command also specifies whether the traversal continues until a null pointer is reached or until a certain number of nodes are traversed. To capture some non-deterministic (input dependent) traversals, there is support to end a traversal on satisfying a simple comparison operator on data in the traversed node and/or a specified constant (elaborated in Section 3.4).

Tree traversals are more varied, and require knowledge of the offsets into a node for the child pointers. We support depth-first (Figure 3(b)), breadth-first, and some non-deterministic traversals. The latter use simple comparison operators on node data and/or constants to determine which successor pointer to follow at each LDS node.

For backbone-and-ribs structures, simple information about the offsets into a node for the "next" backbone pointer and for the ribs needs to be specified, analogous to lists (Figure 3(c)).

3.4 A Prefetch Command

We created and implemented a prefetch command that encodes the LDS traversals described in Section 3.3. As discussed in Section 3.2, the command is not intended to be universally applicable, but to strike a good balance between flexibility and prefetch engine complexity. Further, it is at least as (and in some cases more) flexible as previous schemes for LDS prefetching (Sections 4.2 and 6).

The command consists of the op code and the address of the first node of the LDS to be traversed (as in conventional software prefetching). It is augmented with a two word (64 bits) description of the LDS and traversal path. The command is stored in a register and passed to a prefetch engine

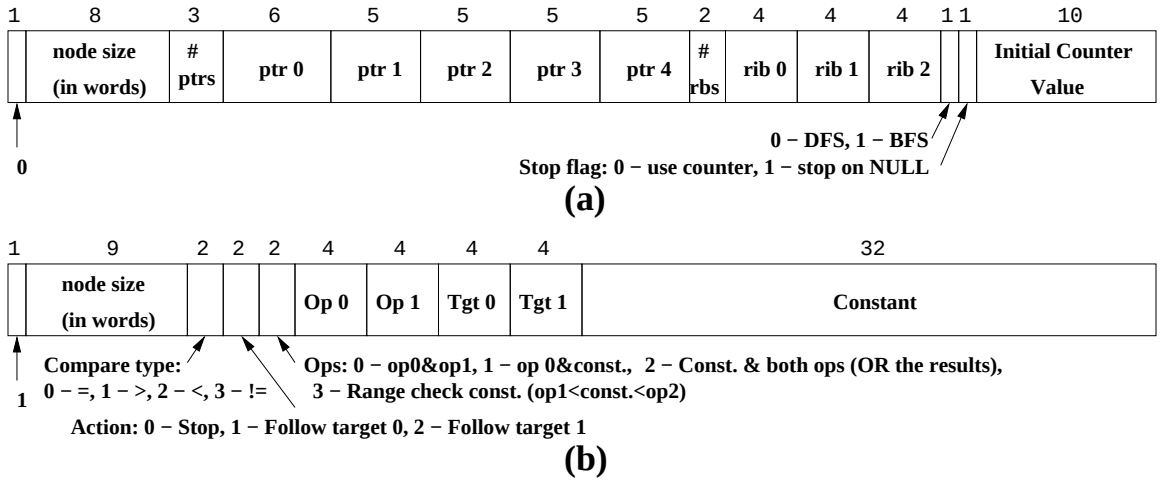


Figure 4: **The format of a prefetch command.** (a) Format for deterministic traversals. (b) Format for input-dependent traversals. The numbers above each field denote the number of bits in that field.

via a new instruction (a special flavor of the store instruction is used in our experiments). There are two types of descriptions – one for deterministic traversal paths (Figure 4(a)) and one for paths with a dependence on the LDS data (Figure 4(b)). Both types contain the size of a node in words.

The command for deterministic traversals includes pointer and rib fields, which are the offsets into a node for the successor pointers and rib pointers. Lists use a single successor pointer while trees use multiple successor pointers. The command also holds the number of valid successor and rib pointers in each node (maximum of five successors and three ribs). For trees, the order of traversal, depth-first (DFS) or breadth-first (BFS), is indicated by a single bit field. Finally, the stop flag instructs the prefetch engine on when to end a traversal. When one, the entire LDS is prefetched; otherwise, a given number of nodes (specified in the initial counter value field) are prefetched.

The other type of prefetch command uses the result of one or two comparisons involving LDS data to determine the traversal path (Figure 4(b)). The compare type field specifies the comparison operation to perform. The operand field specifies the operands to compare (from two fields from the current LDS node and a 32-bit constant). The action field specifies the action to perform on a successful comparison – either to stop the traversal or to continue by following one of the possible successor pointers. The default action (on an unsuccessful comparison) is to follow the first target pointer. The operand and target fields are offsets into the current LDS node and specify the comparison operands and possible successor pointers (targets) respectively. The constant field

holds a full 32-bit constant for use in comparisons. Using 32-bits allows pointer comparisons which are useful when searching for specific nodes in an LDS.

3.5 Executing a Traversal

A processor issuing a prefetch command sends it to the directory of the node that contains the first LDS node to be traversed. The actions required to process the command and the various components of the prefetch engine are illustrated in Figure 2(c) and discussed next.

(1) The directory forwards the command to its prefetch engine. (2) The prefetch engine issues a prefetch to the directory for the first address associated with the request. Prefetches are given lower priority than demand requests, but are otherwise treated like demands until the data is retrieved from the DRAM. (3) In addition to sending a prefetch reply to the requesting processor, the directory sends a copy of the data to the prefetch engine. If the line returned to the prefetch engine is not the last line for the current LDS node, additional prefetches are issued. (4) When the last line for a node returns, the prefetch engine tests if the traversal is complete. If not, it extracts the successor and rib pointers from the LDS node, possibly performing comparisons to determine which successor to follow. (5) The pointers are placed into the address buffer, which orders addresses appropriately for the traversal type (e.g., LIFO for depth-first tree traversals). (6) The next LDS pointer is then removed from the address buffer and (7) sent to the TLB for address translation. The address translation is fast because the prefetch engine has access to the TLB on its PIM chip, as discussed further in Section 3.9. (8) The engine then issues a prefetch for the LDS node. Other, independent pointers (e.g., rib pointers for the last node) can also be prefetched at this time, concurrently with the current node (using steps 6, 7, and 8). Steps 3 through 8 repeat until the traversal terminates (step (4)). The prefetch engine then sends a completion response to the requesting processor via the directory.

3.6 Command Migration – Dealing with Data Distributed on Multiple PIMs

The entire LDS to be traversed is not guaranteed to be present in a single PIM's memory, making it important to run the prefetch command on different PIMs. For a list, this is relatively straightforward. If the next node in a list is on a different PIM, then we *migrate* the execution of the rest of the prefetch command to that PIM. The requesting processor does not have to know that its request has moved. Migrating a tree traversal is more complex since it requires some state information (i.e., the contents of the address buffer) to order the nodes. Multiple policies are possible.

Tree traversals requiring migrations, however, are not critical for any of the applications studied. We, therefore, implemented a simple policy – a request for a cache line on another PIM is migrated only when there is no remaining state; otherwise, it is discarded.

3.7 Cache-Coherence Issues

Memory-initiated communication can introduce new race conditions in a cache-coherence protocol. To simplify a number of cases, we pursued only read prefetching. The use of a relaxed consistency model hides write latency.

The key new race condition introduced is when a directory receives a request for data that it believes is already at the requesting processor. This is not an entirely new case for coherence hardware, but can be caused by two new situations: (1) a prefetch for data that has already been read by another prefetch or demand access by the same processor, and (2) a demand access for data that has just been prefetched by the same processor.

In the first case, the prefetch can simply be squashed. The prefetch engine still needs access to the data to continue the traversal, but the data is not returned to the processor. Since the processor does not notify the directory on cache evictions, this could lead to unnecessary squashing and reduced prefetch coverage. However, we observed that the increase in network traffic due to sending the unnecessary data outweighs the benefits of higher coverage.

In the second case, we send all such demands back to the cache to check if they were unnecessary. This technique will increase the response time for demands that overtake writebacks, but that is typically an infrequent occurrence.

3.8 Coalescing Demand Requests with Prefetches

Traditional prefetches (to individual cache lines [30]) that are sent to memory occupy miss status handling registers (MSHRs) [25], or an equivalent resource, in the processor's cache. If a traditional prefetch is late (returns after the processor requests the data), then any demand access to the same line by the processor will coalesce with the prefetch in the MSHR. This prevents redundant demands from being sent to memory and enables the prefetch to hide part of the latency that would have been seen by the demand. In our system described so far, if a prefetch is late, a redundant demand will be sent out because the cache does not know which lines are being prefetched. Adding a prefetch buffer is not sufficient to prevent redundant demands from being sent. Instead, we add the following hardware to avoid redundant demands.

We expand the MSHRs to have space to hold prefetch command information. We reserve an MSHR for each outgoing prefetch command and use a unique identifier to match prefetch responses with MSHRs when they return. We place address generation hardware analogous to the prefetch engine's hardware next to the MSHRs. The hardware, called the *predictor hardware*, uses the node data returned by a prefetch to predict the next line that will be sent by the prefetch engine. This prediction is used to coalesce a subsequent demand request for the predicted line with the prefetch command in the MSHR. We are, however, not guaranteed that the predictor hardware will predict correctly because the prefetch engine will not return a line if the directory's state shows it as already being in the processor's cache (as discussed in Section 3.7). If a demand request coalesces with a prefetch command and the next line returned is not the same as the predicted one or the traversal is completed, then the cache recognizes that it mispredicted and sends the coalesced demand request down (albeit delayed). An alternate scheme to avoid using predictor hardware is for the cache to interpret a prefetch response as a response to any outstanding demand request to the same line. However, a useless demand request and reply would be generated for each late prefetch, increasing system contention. We found this contention overcomes the benefits of this scheme.

3.9 Support for Address Translation

The prefetch engine and predictor hardware at the cache dereference pointers, which requires a virtual to physical address translation. We use the processor data TLB on the same PIM for this purpose, but always give priority to the processor. We assume a hardware DTLB miss handler as in Intel's Pentium family of processors [18]. We do not increase the size or the ports on the DTLB for our scheme. Thus, its use by the prefetch hardware could increase contention and misses; both effects are modeled in our simulations.

3.10 Support for Avoiding Cache Pollution and Throttling Prefetch Rate

The prefetch engine could potentially hurt performance by causing cache pollution in at least two ways. First, for traversal paths that are not captured exactly by our prefetch command, useless LDS nodes may be prefetched. All current LDS prefetching schemes must deal with this since they cannot exactly capture all traversals. Second, the prefetch traversal may get too far ahead of the processor, replacing useful data in the processor's cache. The memory-side scheme of [40] must deal with this as well. For each of the above cases, cache pollution could be eliminated by providing a prefetch buffer (that is exposed to the cache-coherence protocol), and depositing prefetched data

into this buffer rather than the cache. Additionally, for the second case, to throttle the prefetch rate, we added a delay field to the prefetch command to specify a number of cycles the prefetch engine should wait between issuing prefetches. The user or compiler could determine the delay using an estimate of the work in each iteration. The delay should be $c - (d + m)$, where c is the work per iteration, d is the DRAM latency, and m is the mean prefetch command migration time per access. However, none of our applications benefitted from either the prefetch buffer or the throttling (Section 5.2.3), and so the experiments reported here do not include these capabilities.

4 Experimental ethodology

4.1 Evaluation Environment and Architectures Modeled

We modified the RSIM simulator [33] to model the proposed memory-side prefetching (*MPF*) scheme in Section 3. For comparison, we simulated a *base* system that is identical to MPF, but without the prefetch engine. We also simulated software jump-pointer based processor-side prefetching (*PPF*) as discussed in Section 2. The PPF system is identical to the base system and additionally supports a conventional software prefetch instruction. We did not add a prefetch buffer to the PPF system as in [36] because we found the benchmarks exhibited little cache pollution due to prefetching (Section 5.2.2). Also, the buffer would need to be visible to the coherence mechanism since we model a multiprocessor system; the system in [36] is a uniprocessor where this extra complexity is not required.

Table 1 summarizes the system parameters for the base architecture. Current PIM chips contain relatively simple processors. In the future, with the increasing number of on-chip transistors, it would be possible to implement a current state-of-the-art processor and a reasonable amount of DRAM on the same chip. We therefore chose to model a current state-of-the-art superscalar out-of-order processor for our study.

Since the instruction footprint of our benchmarks is small, we assume that all instructions hit in the instruction cache and in the instruction TLB (with a single cycle hit time). The data cache size chosen is sufficient to hold the first-level working sets for all benchmarks, but not large enough to hold their second-level working sets (as would be expected in several realistic scenarios). This follows the methodology of Woo et al. [39] which suggests scaling down the data cache sizes based on application input sizes (which are typically scaled down for simulation). The latency gap between an L2 cache and main memory is significantly lower when main memory is on chip, making

Memory Hierarchy and Network Parameters		ILP Processor	
L1 D-cache (on-chip)	64K, 2-way associative,	Processor Speed	600MHz
	64B line, 2 ports, 8 MSHRs	Fetch/Retire Rate	4 per cycle
DTLB	128 entries, fully associative,	Functional Units	2 Int, 2 FP, 2 Add. gen.
	hardware-managed,	FU Latencies	1/3/9 int. add/mult./div.
	2 ports, 30 cycle miss penalty		3/4/10 FP add/mult./div.
Bus (on-chip)	600 MHz, 128 bits, split trans.	Instruction window	64 entries
Memory (on-chip)	4-way interleaved, 30ns	(reorder buffer) size	
	access, 16B/cycle	Memory queue size	32 entries
Network	2D mesh, 64 bits,	**Contentionless Memory Latencies**	
	4 cycle flit delay per hop	L1 hit time (on-chip)	1 cycle
PIMs in system	16	Local Memory (on-chip)	26 cycles
		Remote Memory	90-170 cycles
		Cache-to-cache transfer	108-201 cycles

Table 1: **Parameters for the base system**

an L2 cache on a PIM less cost-effective. Therefore, our main results model only a single level of cache. We also simulated architectures with an L2 cache (including cases where the second-level working set may fit in the L2 cache) and discuss those results in Section 5.4.

We also performed experiments that assess the sensitivity of our results to memory latency and discuss them in Section 5.4.

4.2 Evaluation Workload

We use the Olden benchmark suite [5], the collection of pointer-intensive codes most commonly used in recent LDS prefetching studies [27, 36, 40]. We present results for seven of the ten Olden benchmarks; the other three either have very small memory stall time ($< 10\%$ for *barnes* and *power*) or are dominated by dereferencing of computed addresses rather than pointers (in *voronoi*). Table 2 provides a brief description of the benchmarks studied and the input parameters used. (The last column is discussed below.)

The Olden source codes contain parallelization hints for a parallelizing compiler. We manually parallelized four of them, *em3d*, *health*, *perimeter*, and *treeadd*, faithfully following these hints. The other three benchmarks, *bisort*, *mst*, and *tsp*, do not scale well when parallelized, so we leave them single-threaded. All benchmarks are run on a system with 16 PIMs. The multithreaded benchmarks use all 16 processors. If possible, their LDS nodes are placed on the PIM that is most

Benchmark	Description	LDS prefetched	Input data size	PPF Idiom
bisort	Performs ascending and descending bitonic sorts	Dynamic binary tree	64K nodes	queue
em3d	Simulates propagation of EM waves in a 3D body	Static linked list with ribs	4K H nodes 4K E nodes	staggered chain
health	Simulation of the Columbian health care system	Dynamic linked lists	level=5 time=300	full
mst	Builds a minimum spanning tree	Static linked lists	1024 nodes	root
perimeter	Computes the perimeter of regions in images	Static four-way tree	1K x 1K image	queue
treeadd	Sums the values in a tree	Static binary tree	1M nodes	queue
tsp	Traveling salesman problem	Dynamic linked lists	64K nodes	queue

Table 2: **Benchmark characteristics**

likely to traverse them. (This may not always be possible since data placement in the simulated system is done at the granularity of a page.) The single-threaded benchmarks use one processor (including for MPF and PPF), but have their data (randomly) distributed amongst the 16 PIMs at page granularity (as would be the case if the data set were too large to fit in a single PIM's memory). MPF uses the prefetch engines on all 16 PIMs for both parallelized and single-threaded applications.

Prefetches were inserted by hand for both MPF and PPF. We do not use a compiler to insert the prefetches because, to our knowledge, no current compiler is capable of inserting prefetches for either MPF or PPF. To ensure a fair comparison, we obtained the code used in [36] to determine where to insert prefetches for PPF and used analogous prefetches for MPF. Except for *bisort* and *perimeter*, the use of the appropriate jump-pointer idiom for PPF and the prefetch command for MPF was relatively straightforward for the primary LDS traversals. *Bisort* and *perimeter* make multiple passes through their primary LDS, but traverse only part of the LDS each time. *Bisort* uses comparisons involving dynamic data from multiple nodes to determine traversal paths. *Perimeter* traverses up and down an unbalanced binary tree, where the traversal path is dependent upon the shape of the tree. PPF cannot build jump pointers to capture these traversals precisely and our MPF prefetch command cannot capture them precisely either. While it is possible to attempt to prefetch along all possible paths for these two benchmarks (as in [21]), bandwidth can become a limitation and degrade performance. Therefore, prefetches are issued for only a very small number

of the LDS node accesses in these cases for both PPF and MPF.

For PPF, when multiple jump-pointer prefetching idioms were applicable to the same benchmark, we selected the idiom that provided the best performance on our architecture for that benchmark as summarized in the last column in Table 2. We also adjusted the jump intervals to achieve maximum performance.

For MPF, *mst* and *tsp* use the non-deterministic versions of the prefetch command. *Mst* traversals examine hash table buckets, terminating when a node with the matching key is found. *Tsp* traversals examine circularly linked lists, terminating when a successor pointer is the same as the first node in the list. PPF does not use separate support for these traversals since its prefetches are issued and terminated synchronously with the processor's traversal. As a result, in *mst* some root-jumping traversals may be terminated a little too early and some may result in a few extra nodes being prefetched. For all benchmarks, we attempted to minimize the number of useless prefetches issued. At times, this forced us to place the prefetch commands in MPF very close to the first access to the LDS.

All benchmarks except *health* contain a large initialization phase where the data structures are built. We start our measurements after this phase and after a warm-up time for *health*.

4.3 Evaluation Metrics

The primary metric used in our evaluation is execution time. For further insight, execution time is divided into six components – busy time, functional unit stall time, local memory stall time (stall time for memory accesses resolved within the local PIM, either at the L1 cache or local DRAM), remote memory stall time (stall time due to remote memory accesses and cache-to-cache transfers), TLB miss stall time (stall time waiting for the TLB miss handler to complete), and synchronization stall time. Busy and stall times are calculated similar to previous work [32]. For each cycle, the fraction of instructions retired relative to the maximum retire rate is recorded as busy time. The remaining fraction of the cycle is charged as stall time for the first instruction in the instruction window unable to retire; however, if the first instruction is waiting for a TLB miss to be resolved, the time is charged as TLB stall time.

We also report statistics on prefetch coverage and the fraction of prefetch data transfers (as opposed to aggregate prefetch commands) that are useless, late, and damaging as follows. We measure *prefetch coverage* as the reduction in total (including non-LDS) demand read miss requests. A prefetch data transfer is *useless* if the data it returns is not used by the processor before being

evicted. If the predictor hardware at the cache mispredicts and sends a demand request for a line for which a prefetch is already on its way back from memory, then the corresponding prefetch data transfer is also counted as useless. A *late* prefetch data transfer is a prefetch that is not useless, but arrives after the corresponding demand access, thereby exposing some part of the memory latency. A prefetch data transfer is *damaging* if it replaces a line that is needed by a subsequent demand access.

5 Results

This section presents our results. Section 5.1 presents a qualitative framework to understand the benefits and limitations of our MPF scheme and jump-pointer PPF and determine when either would be best. Section 5.2 presents quantitative results on the benefits of our MPF scheme over the base architecture. Section 5.3 presents quantitative results comparing MPF and PPF. Section 5.4 discusses the applicability of our results to other system architectures.

5.1 Qualitative Framework

We identify three key factors that determine the performance benefits of our MPF scheme and jump-pointer PPF. We then use the analysis to summarize when each scheme would be best.

Instruction overhead

MPF requires a single prefetch command for the entire LDS traversal; consequently, instruction overhead is likely to be negligible. In contrast, PPF requires an additional instruction for each LDS node prefetched and for the creation and possibly updating of jump-pointers. This overhead can be considerable for dynamic structures that require frequent updates. Staggered chain jumping incurs overhead as well because delaying chained prefetches is similar to updating jump-pointers.

Unoverlapped latency in the startup period

Both MPF and PPF incur a *startup period* where the latency of the initial nodes in the LDS is exposed; i.e., the prefetched data arrives later than the demand access for these nodes. MPF's startup period is typically small. If the prefetch command is issued just before the LDS traversal, then accessing the very first node will require waiting for a round trip time to memory. Subsequent node accesses are part of the steady state which is analyzed later. If the prefetch command can be issued sufficiently early, then even the first node of the LDS will not incur stall time.

The startup period for PPF extends to the first jump interval since nodes in this period do

not have jump pointers. For all jump-pointer idioms other than root jumping, the jump interval is a function of the computation per node and the memory latency. For root jumping, the startup period covers work on the entire first LDS (since none of its nodes are prefetched); therefore, it is a function of the LDS length and memory latency.

To better quantify the above effects, we present a simple "first-order" analytical model of the memory stall time incurred during the startup period. Let l be the average memory latency seen at the processor, L be the length of the LDS, c be the computation performed per node, and e be the time between the issue of a prefetch command and the first LDS demand access (denoting how *e*arly the command is issued).

For MPF, $Startup\ stall\ time \approx max\{0, l - e\}$

For queue, full, and chain jumping, $Startup\ stall\ time \approx jump\ interval \times l$, where

$jump\ interval = \lceil \frac{l}{c} \rceil$ for queue, full, and regular chain jumping

$jump\ interval = \lceil \frac{2l}{c} \rceil$ for staggered chain jumping

For root jumping, $Startup\ stall\ time \approx L \times l$

The above equations show that for MPF, the startup stall time is at most equal to the memory latency and can be zero if the prefetch command can be issued early enough. In contrast, for PPF, the startup cost can be considerable – it is a function of the square of the memory latency for all but root jumping.

Steady state prefetch behavior

After the startup period, both MPF and PPF enter a kind of steady state, where the processor may spend a certain amount of time waiting for a node to load (if the prefetched data arrives late) and then works on the node. In MPF, typically there is a delay (say d) of approximately one DRAM access time between two prefetches issued by the prefetch engine (d is 26 cycles in our system). Assuming d is the longest stage in the entire (pipelined) path of a prefetch request and transfer and there are no prefetch command migrations, prefetched data will appear at the processor every d cycles. With migration, the average migration time per node, say m, is added to this delay (m differs across applications, and depends on the network latencies and the number of migrations per node). If the computation time per node, c, is more than d+m, the prefetched data arrives before the demand access and there is no steady state stall time. If c is less than d+m, then each node sees a stall time of d+m-c unless the prefetch command is issued early enough. In the latter case, some initial nodes of the LDS may have been prefetched before their demand accesses and

will see no stall time. In this case, the post-startup time actually consists of two phases (with the first phase incurring no late prefetches), but we continue to refer to it as the "steady state" for simplicity. Thus, for MPF

if $c \geq d+m$,

Avg. steady state stall time per node ≈ 0

if $c < d+m$,

Avg. steady state stall time per node ≈ 0 for a few initial nodes of the LDS

Avg. steady state stall time per node $\approx d+m-c$ for the remaining nodes

For PPF, with queue, full, and staggered chain jumping, the jump interval can be adjusted to match the amount of work done per LDS node such that there are no late prefetches and no memory stall time. Root jumping does not have this advantage because it prefetches nodes without using a jump interval. Instead, the next LDS to be accessed is always prefetched in lockstep with the current one, allowing overlap with the work for only a single node. An analogous observation applies to regular chain jumping as well. Thus,

For queue, full, and staggered chain jumping, *Avg. steady state stall time per node* ≈ 0

For root and regular chain jumping, *Avg. steady state stall time per node* $\approx max\{0, l-c\}$

In summary, MPF sees no steady-state stall time if the computation time per node is more than the sum of (1) the memory latency seen by the prefetch engine (approximately the DRAM latency in our case) and (2) the average per-node prefetch command migration time. Otherwise, MPF could see some stall time unless the prefetch command is issued early enough. Prefetch commands are more likely to be issued early enough to eliminate the stall time if the LDS is small. Full, queue, and staggered chain jumping do not see steady state stall time. Root and regular chain jumping see steady state stall time unless computation time per node is more than the round-trip memory latency (seen at the processor).

Other issues

The above three factors were dominant in our experiments. The following could become important in some situations, favoring either MPF or PPF depending on additional support provided.

The *bandwidth* requirement for MPF could be higher or lower than for the base case, depending on the number of LDS nodes traversed per prefetch command, missed coalescing opportunities,

prefetch command migrations, and useless prefetches. PPF, on the other hand, cannot reduce bandwidth, but may increase it if there are useless prefetches.

Cache pollution is a potential problem for both schemes. The problem and potential solutions were discussed in Section 3.10.

Our MPF scheme moves data from DRAM into the cache hierarchy, but cannot hide the latency of *hits in the L2 or lower levels of the cache hierarchy.* Yang and Lebeck's memory-side scheme replicates the prefetch engine at each level in the memory hierarchy to address this issue [40]. We found this unnecessary for the applications we studied (Section 5.4), but could modify our scheme accordingly if hiding lower level cache latency becomes important. PPF, on the other hand, could potentially hide these latencies in a natural way. However, our experiments (Section 5.4) showed that in some cases the instruction overhead incurred by PPF offsets any potential benefits. To our knowledge, current compiler prefetching algorithms schedule prefetches for main memory latency rather than L2 latencies since the former remain important even with large L2 caches (e.g., the Pentium 4 and Itanium processors provide options to prefetch into lower cache levels, underlying the importance of misses that need to go to main memory [17, 16]). L2 cache latencies can also be handled with other techniques such as non-blocking loads and out-of-order processors.

Finally, jump-pointer PPF is potentially more flexible than our MPF scheme, and could capture LDS traversals that MPF cannot. However, for our benchmarks this was not the case. Also, jump-pointers need to be assigned, requiring a repeated traversal of the LDS, whereas our MPF scheme does not.

Summary

Table 3 summarizes the above analysis. MPF is expected to be better when the LDS being accessed is small (relative to the jump interval required), while PPF is better when the LDS accessed is large and has little work done per node (relative to the memory latency seen by the prefetch engine and command migration overhead). For large LDS with a reasonably large amount of work done per node, the startup and steady state effects are negligible for both schemes. However, the instruction overhead of PPF makes MPF more attractive, especially if the LDS is dynamic or requires many prefetches per node (e.g., backbone and ribs structure).

5.2 Impact of Memory-Side Prefetching on the Base Architecture

For each benchmark, Figure 5 shows the normalized execution times for the base system without prefetching (Base), the base system with memory-side prefetching (MPF), and the base system

Scheme	Advantages	Where is it the best scheme?
MPF	(1)Lower instruction overhead (2)Shorter startup time	(1) Small LDS (2) Large LDS with lot of work per node, where LDS is dynamic or requires many prefetches per node
PPF (Software jump-pointer prefetching)	(1) Shorter steady state stall time, except for root or regular chain jumping	Large LDS with little work per node

Table 3: **Characterization and comparison of MPF and PPF.**

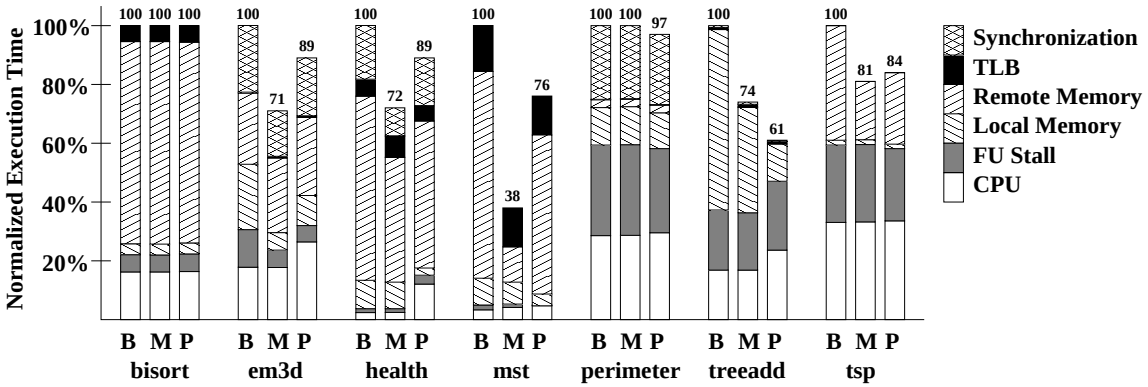


Figure 5: **Normalized execution times.** Results for base system (B), memory-side prefetching (M), and processor-side (software jump-pointer) prefetching (P).

with processor-side prefetching (PPF). The bars are normalized to the time for Base for the corresponding benchmark. Each bar is split into the busy and stall components of execution time as described in Section 4.3. This section focuses on the Base and MPF bars.

5.2.1 Overall Results

Figure 5 shows that five of the benchmarks see significant improvements in performance from MPF (19% to 62%), while the other two, *bisort* and *perimeter*, see an insignificant change. The average reduction in execution time over the six benchmarks with significant LDS stall time (i.e., excluding *perimeter*) is 27%.

Bisort and *perimeter* do not see any benefits because their traversal paths cannot be exactly captured by the implemented prefetch command and so very few prefetches could be inserted, as

App.	Coverage	Local		Remote		% of Prefetch Transfers			%Miss	Nodes/	Mig./
		% in base	%Reduced	% in base	%Reduced	Useless	Late	Dam.	Coal.	Cmd	Node
em3d	84.3	94.5	88.5	5.5	11.9	3.1	4.9	2.1	1.0	256.0	0.0
health	53.5	37.4	15.0	62.6	76.5	10.9	71.0	4.3	8.5	57.4	0.89
mst	57.0	45.9	7.4	54.1	99.0	0.1	61.5	0.0	0.0	3.0	0.96
treeadd	67.9	100.0	67.9	0.0	N/A	26.5	0.2	0.2	26.5	65535.0	0.0
tsp	70.1	7.5	68.7	92.5	70.2	2.4	96.2	1.6	0.2	668.0	0.11

Table 4: **Prefetch coverage and statistics.** The first group of columns shows the prefetch coverage (overall and broken down into local and remote). The second group shows useless, late, and damaging prefetches. The final group shows some statistics about the prefetching hardware's performance (% of missed coalescing opportunities, LDS nodes traversed per prefetch command, and average number of migrations per LDS node traversed).

discussed in Section 4.2. The traversal paths of the other five benchmarks can be captured by the implemented prefetch command, and they see significant improvements. The next section provides a more detailed analysis of the causes of the benefits and remaining limitations of MPF for these five benchmarks.

5.2.2 Detailed Analysis

For a more detailed analysis, we examine prefetch coverage and the fraction of prefetch data transfers (as opposed to aggregate prefetch commands) that are useless, late, and damaging (Table 4). (These terms are explained in Section 4.3.) Additionally, Table 4 provides supplemental data to explain some of the above prefetch statistics. Our analysis draws on the framework discussed in Section 5.1.

Prefetch Coverage

The first part of Table 4 gives the reduction in all demand read miss requests (prefetch coverage), as well as a breakdown of these requests into local and remote reads. We see that all five benchmarks have high prefetch coverage (over 50% for all benchmarks), and both local and remote reads are reduced. The reason that the coverage is not higher is that there are a significant number of non-LDS read misses (they could potentially be prefetched with more conventional prefetching [30]), and a few LDS misses that could not be captured by our command (in *tsp*). Also, for *health* and *treeadd*, sometimes the predictor hardware at the cache mispredicts, making some prefetches useless and reducing the coverage, as elaborated next.

Useless Prefetch Data Transfers

The second group of columns in Table 4 shows that the fraction of prefetch data transfers that are useless is small for all five benchmarks, and under 5% for *em3d*, *mst*, and *tsp*. The reason for the relatively higher number of useless prefetches for *health* and *treeadd* is explained by the "Miss Coal" column in the table. This column contains the fraction of prefetch data transfers that could have had a corresponding demand request coalesce with their prefetch command, but did not because of a misprediction by the predictor hardware. In these cases the data transferred by the prefetch is not used. The *health* and *treeadd* benchmarks have a significant number of missed coalescing opportunities, explaining the higher fraction of useless prefetch transfers (and relatively lower prefetch coverage). For *health*, mispredictions are caused by a small amount of data reuse. For *treeadd*, they arise because the computation time per LDS node is small relative to the DRAM latency. As discussed in Section 5.1, this implies that the prefetched data does not arrive at the processor before its corresponding demand access, incurring steady state stall time. Such prefetches would usually be seen as late prefetches. However, in *treeadd*, the processor can send out multiple requests in parallel when traversing leaves of the tree. Since only one request can coalesce with a prefetch command at the cache, the others suffer mispredictions and incur useless prefetches and steady state stall time.

Late Prefetch Data Transfers

Table 4 shows a significant number of late prefetch data transfers for *health*, *mst*, and *tsp*. Based on the framework in Section 5.1, prefetches can be late if computation time per node is too little or if migration time per node is high. These effects, however, can be mitigated for short LDS if the prefetch command is scheduled significantly earlier than the LDS access.

Health and *mst* have small computation time per node, but are able to send prefetch commands early and have short LDS (second to last column of Table 4). The cause for late prefetches in these benchmarks is the high prefetch command migrations (last column of Table 4).

Tsp sees late prefetches because it has relatively small computation time per node and is unable to send out prefetch commands early enough for its long LDS. This benchmark also sees some migrations, further contributing to the late prefetches.

Damaging Prefetch Data Transfers

Overall, the number of damaging prefetches is very low for all five benchmarks (less than 5% as shown in Table 4). The number of damaging prefetches is largely dependent on the number of prefetch transfers that return too early or that are useless. The early prefetches result from two

factors – the placement of the prefetch commands in the code and the computation per LDS node. First, if a prefetch command is issued well before any use of the LDS data, then prefetch transfers may return too early and cause the eviction of useful data. This is the cause of the 4% damaging prefetch data transfers in *health*. There is a balance in *health* between prefetching some data too early and placing the prefetch command too late; the implementation used provided optimal performance. Second, if a processor performs a lot of work on some nodes, then the prefetch engine may get too far ahead of it and return some data too early. This is the case in *em3d* and *tsp*. (*Mst* and *treeadd* have negligible damaging prefetches.) Nevertheless, in all cases, the number of damaging prefetches is low enough to not have any significant impact on performance.

Summary

In summary, our prefetch command is able to capture most LDS traversals in our benchmark suite. For the benchmarks with such traversals, MPF is effective at reducing both local and remote memory stall time. The prefetch coverage is fairly high, and the number of useless prefetches issued is relatively small since there is little speculation involved. The performance benefits are primarily limited by the DRAM latency and migration rate relative to the work done per node. This is manifested as late prefetches for *health*, *mst*, and *tsp*. For *treeadd*, this is instead manifested as useless prefetches. Note that in *em3d* and *health*, synchronization stall time is also reduced, due to a reduction in load imbalance. In all benchmarks, some of the remaining latency is due to non-LDS accesses, as they were not addressed in this paper. Finally, damaging or early prefetches were not a limiting factor in our benchmarks.

5.2.3 Evaluation of Features of the Prefetching Hardware

The migrations/node column of Table 4 shows that support for command migrations is critical (more quantitative justification appears in [13]). The data on late prefetch transfers (Table 4) shows that support for coalescing demand misses with prefetch commands is also critical. The low number of damaging prefetch transfers shows negligible cache pollution for these benchmarks. If cache pollution does become a problem, then a prefetch buffer and delay fields in the prefetch command would need to be used (Section 3.10). Finally, Figure 5 shows that using the processor data TLB for the prefetch engine has an insignificant impact.

5.3 Comparison of Memory-Side and Processor-Side LDS Prefetching

This section compares our MPF scheme to jump-pointer PPF. From Figure 5, we see that MPF significantly outperforms PPF for *em3d*, *health*, and *mst*, with a reduction in execution time of 20%, 20%, and 50%, respectively. PPF significantly outperforms MPF for *treeadd* with a reduction in execution time of 18%. Both schemes perform similarly for *bisort*, *perimeter*, and *tsp* (within 4%). The following uses the framework from Section 5.1 to explain these results.

For *em3d*, the LDS are of moderate size with a large amount of work done per node, so neither technique has significant steady state stall time. MPF performs better because PPF incurs large instruction overhead from staggered chain jumping and a larger startup period.

The LDS used in *health* are dynamic, relatively small, and the work per node is small. The primary reasons for PPF's poorer performance are its significant startup period and its overhead of maintaining the jump-pointers for the dynamic LDS. MPF does suffer from some steady state stall time due to the small amount of work done per node, but prefetch commands for the LDS can be issued early, hiding most of it.

Mst accesses many hash table buckets, so the LDS are very small and very little work is done per node. The structures are static and root jumping is used for PPF, so instruction overhead and startup effects are negligible. Root jumping, however, suffers from steady state stalls. MPF, on the other hand, can send prefetch commands early enough, largely eliminating steady state stalls for these small LDS and outperforming PPF.

For *treeadd*, the LDS accessed are very large, static binary trees which have little work performed per node. For PPF, the startup period and instruction overhead are quite small, and its use of queue jumping ideally incurs no steady state stalls. MPF has significant steady state stalls because of the little work done on each node and the large size of the LDS, thereby underperforming PPF.

Tsp's LDS are large, static, and have a moderate amount of work done per node. Both MPF and PPF therefore perform similarly and reasonably well.

Finally, neither technique works well for the non-deterministic traversals of *bisort* and *perimeter*.

In summary, we find that our MPF scheme is effective, but a combination of jump-pointer PPF and our MPF scheme is best. The third column of Table 3 can be used to decide which scheme to use for a specific LDS traversal.

5.4 Alternative Architectures

Non-PIM architectures. In this work, we have chosen a system composed entirely of PIMs for our evaluations. Our fundamental insights from Section 5.1 on when MPF or PPF is better, however, are also applicable to alternative configurations. For example, the MPF prefetch engine (and a TLB-like structure) could be integrated with the memory or directory controller which could be on a different chip than the DRAM (as assumed in [40]), or the prefetch engine and DRAM may be integrated but the main processor may be on another chip. In each case, the system could be uniprocessor or multiprocessor. The impact of the configuration is manifested in two key parameters identified in Section 5.1 – the values of the round-trip memory latencies as seen at the prefetch engine (d) and at the processor (l). The farther the prefetch engine from the memory (larger d), the larger the potential steady state stall time for MPF. The farther the main processor from the memory (larger l), the larger the potential startup stall time for both MPF and PPF. However, the startup time varies as l^2 for most PPF schemes but only as l for MPF, and so MPF is likely to benefit more with large l (for constant d).

As one additional data point, we performed experiments doubling all the latencies in the memory hierarchy in the base system (i.e., doubling d and l). The results were qualitatively the same as those with the base latencies. Quantitatively, we found that for all benchmarks except for *treeadd*, there is very little change ($< 6\%$) in the relative performance difference among all systems evaluated. For *treeadd*, which is the only benchmark where PPF significantly outperformed MPF, the difference in the two schemes widened further (from 18% to 36%). These results can be explained using our framework as with the original system. Further details on these results can be found in [14]. A more extensive exploration of the architectural design space is outside the scope of this work.

L2 caches. To assess the impact of a deeper cache hierarchy, we evaluated MPF and PPF on the base system augmented with a 1 MB L2 cache with a latency of one-half of the base local memory latency. There were two benchmarks – *health* and *tsp* – where a much larger part of the working set fit in this cache. The resulting reduction in memory stall time in the base system decreased the relative benefits from prefetching, but magnified the overheads (instruction overhead for PPF and TLB overhead for MPF). Consequently, for these benchmarks, both PPF and MPF gave insignificant benefits ($< 10\%$) and MPF degraded performance for *health*. For the other benchmarks, the relation between PPF and MPF stayed the same as before. MPF is better than PPF for *em3d* (by 12%) and *mst* (by 52%). PPF is better than MPF for *treeadd* (by 31%). (In

these experiments, we favored PPF because we prefetched into the L1 for PPF and into the L2 for MPF.) Detailed results can be found in [14].

6 elated Work

There has been a large amount of work in prefetching, most of which has focused on regular data structures (e.g., [7, 8, 30, 34]). More recently, a number of prefetching techniques for LDS have been proposed, all but one of which are processor-side prefetching.

Luk et al. proposed using jump-pointers [27], which were further developed by Roth and Sohi [36] as discussed in Sections 2 and 5. This paper has so far focused on a software implementation of the four jump-pointer idioms proposed by Roth and Sohi. They also proposed hardware and cooperative hardware/software implementations that use significant additional hardware support at the processor to overcome some of the software scheme's limitations. The hardware automatically creates and updates jump-pointers and generates addresses for and issues prefetches (including those for root and chain jumping). The hardware can eliminate the instruction overhead of jump pointers and can reduce the steady state stall time for root and chain jumping, but it does not affect the startup stall time for any case and does not eliminate the steady state stall time for root and chain jumping. We therefore expect the memory-side scheme to continue to outperform even the cooperative and hardware jump-pointer prefetching implementations for some applications; however, any comparison between these schemes must also consider hardware complexity. A more detailed analysis explaining the difference between the performance of these schemes is presented in Section 4.2.4 of [13].

Recently, Karlsson et al. proposed a technique to reduce the startup stall time for jump-pointer prefetching [21]. This technique creates an array of pointers, called the prefetch array, pointing to the first few nodes in an LDS that do not have jump-pointers. Just before accessing the LDS, prefetches are issued to all nodes pointed by the prefetch array, potentially hiding some latency for those nodes as well. We implemented prefetch arrays for the relevant benchmarks. With jump-pointer prefetching and prefetch arrays, *health* sees a 20% reduction in execution time over the base system (as opposed to 11% with jump-pointers but without prefetch arrays and 29% with memory-side prefetching). The other benchmarks showed or are expected to show little benefit over jump-pointer prefetching without prefetch arrays or even some performance *degradation*. A more detailed analysis and the quantitative results appear in Section 4.2.5 of [13]. Overall, we

found that the prefetch arrays technique did not give consistent benefits over the jump-pointer prefetching schemes of Roth and Sohi, but has potential for hiding startup stall time in some cases. A more detailed characterization of those cases and a combined approach with memory-side prefetching is a promising direction for future work, but outside the scope of this work.

Kohout et al. have proposed an LDS prefetching scheme that employs a processor-side prefetch engine [23]. The engine performs a serialized traversal of the LDS rather than use jump-pointers. This work develops a prefetch scheduling algorithm to reduce startup stall time by overlapping the startup period with "pre-loop" work. The scheme is shown to outperform queue jumping as proposed by Luk et al. [27]; however, jump-pointer creation overhead in the initialization phases of the applications is also charged against queue jumping and a comparison with the other jump-pointer idioms developed by Roth and Sohi [36] is not provided. We expect a memory-side scheme to outperform the above scheme because prefetching can be initiated just as early for memory-side prefetching and its prefetch engine has shorter round-trip time to memory.

Other processor-side prefetching schemes include SPAID [26], group prefetching [41], greedy prefetching [27], LDS linearization [27], and dependence-based prediction [29, 35]. These schemes are more limited than the jump-pointer schemes (e.g., most serialize LDS prefetches or do not handle truly dynamic LDS).

The only other memory-side prefetching scheme of which we are aware is by Yang and Lebeck [40], developed for a uniprocessor, non-PIM system [40]. They refer to their scheme as the push model, and to processor-side schemes as the pull model of data movement. They propose a prefetch engine at each level of the memory hierarchy to handle linked lists. Our engine is driven by a software command. In Yang and Lebeck's scheme, the processor downloads a kernel of load instructions, which encompass the LDS traversal, to the prefetch engine. The prefetch engine then "executes" the load instructions repeatedly to traverse the LDS. The lack of address ordering hardware (e.g., our address buffer) and comparison hardware restricts their scheme's traversals to linked lists (with or without ribs) and excludes data-dependent traversals. The kernels and prefetch engine would require significant changes to allow more general traversals. They compare their scheme to the dependence-based prediction scheme of Roth et al. [35] (which serializes prefetches), but not the more effective jump-pointer schemes that we have examined. They find that memory-side prefetching always outperforms their processor-side scheme for their macro-benchmarks. Their analysis with a microbenchmark indicates that memory-side prefetching is better with less computation per node. We see different results (e.g., processor-side is better with less computation per

node) since we compare with more aggressive processor-side schemes that allow multiple prefetches to be overlapped. We also develop a qualitative framework for when the processor-side schemes or the memory-side schemes should be used for best performance.

Our memory-side LDS prefetching scheme is at least as flexible (in LDS traversal paths captured) as previous LDS prefetching schemes (processor- or memory-side). Moreover, it is the only one to have an (albeit limited) ability to adapt to input-dependent traversal paths. Compared to the other memory-side scheme [40] (and some of the processor-side schemes), it can additionally handle LDS with multiple successors per node. Compared to jump-pointer schemes, it is better able to handle dynamic LDS.

Researchers have also proposed techniques for improving spatial locality of LDS, including data placement [4], cache-conscious structure layout [9], and memory forwarding [28]. These techniques are either inapplicable to or have limited effects on applications with dynamic structures. Impulse is an intelligent memory controller that remaps physical addresses to improve performance of irregular data access patterns [6]. It implements next-line prefetching and buffers the data at the memory controller. Another class of schemes, called correlation-based schemes, use past access patterns to predict future accesses [1, 2, 19]. Although they are not specifically targeted towards LDS data, they act on those as well. On a request for a cache line, they record the addresses of the next set of requests. When the same line is accessed again, a limited number of previous successors are prefetched. Large structures, therefore, cannot be represented efficiently. The processor-side schemes also increase bandwidth requirements significantly since multiple prefetches may be issued for each line accessed. Finally, these schemes do not attempt to get far enough ahead of the processor to hide all of the memory latency.

There has also been a lot of work on PIMs. Burger et al. developed the DataScalar architecture [3], a multiprocessor PIM system for running uniprocessor applications. The PIMs all run the full application on the entire input data set, but act as intelligent prefetch engines for one another by sending local data to the other processors as it is needed. While this generates excellent prefetching characteristics, the large amount of redundant computation makes inefficient use of the hardware. Many other PIM systems have been proposed, including CRAM [10], Execube [22], the Terasys Massively Parallel PIM Array [11], Saulsbury et al.'s system [37], Vector IRAM [24], Active Pages [31], DIVA [12], and FlexRAM [20]. These systems can be split into two types: (1) the main processor integrated with memory, and (2) co-processors integrated into the system DRAM for data parallel operations. None of these systems includes specific support to accelerate an indi-

vidual LDS traversal (although DIVA reports speedups on LDS codes by performing independent LDS traversals concurrently on multiple PIMs [12]).

7 Conclusions

The ability to quickly traverse linked data structures (LDS) will increase in importance with the popularity of object-oriented programming and application domains that involve large dynamic structures. The inherently serial nature of LDS traversals makes them difficult to optimize. This work studies a memory-side scheme for prefetching LDS that consists of a simple prefetch engine that sits close to memory. The engine receives a prefetch command encoding a summary of the LDS and the traversal from the processor. The engine performs the traversal, potentially running ahead of the processor and pipelining multiple LDS data items to the processor before the demand accesses.

The memory-side scheme is ideally suited to a system that can integrate the processor, memory, and prefetch engine on the same chip, allowing the prefetch engine a fast path to memory and the processor's TLB. Some have speculated that future high performance systems will take the shape of multiprocessor systems built out of such PIM chips. We therefore used such a sytem as the underlying architecture for our design and evaluations.

Using the Olden benchmark suite, we found that for the six benchmarks where LDS memory stall time is significant the memory-side scheme reduces execution time by an average of 27% (range of 0% to 62%) compared to a system without any prefetching. We also compared the memory-side scheme to a state-of-the-art processor-side prefetching scheme based on software jump-pointers. We found that the memory-side scheme outperforms the processor-side scheme for two classes of applications: (1) where the LDS are small compared to the round-trip memory latency (these incur high startup time for processor-side prefetching), and (2) where the LDS is dynamic or requires many prefetches per node (these incur high instruction overhead for processor-side prefetching). The processor-side scheme outperforms the memory-side scheme for large LDS with little work per node. This is because the rate of prefetch transfers for the memory-side scheme is limited by the memory latency seen by the prefetch engine and the prefetch command migration frequency; if this rate is low relative to the work done per LDS node, then memory-side prefetching sees stall times in the steady state. Thus, we conclude that a combination of the memory-side and processor-side schemes examined here would prove most effective as a general technique; our characterization

can aid in choosing the appropriate scheme for a specific LDS. Our results are different from a previously proposed memory-side scheme [40], primarily because we compare with a state-of-the-art processor-side scheme that allows parallel prefetches.

In this work, we have chosen to study a multi-PIM system with a current state-of-the-art processor integrated with DRAM on the PIM. Nevertheless, our framework characterizing the situations where the memory-side scheme or the processor-side scheme is better is applicable to other configurations as well.

The software prefetch command used by the memory-side scheme in this work is not meant to be universally applicable, but can capture most traversals in the Olden suite and provides as much (or more) flexibility than previous LDS prefetching schemes. An interesting avenue of future work is to explore a prefetch engine that can capture a general traversal, but with acceptable hardware complexity. Another promising avenue would be to analyze other pointer-intensive applications, such as OLTP applications, to see what LDS traversal patterns are present and if our engine can capture them as is.

Acknowledgments

We would like to thank Amir Roth for providing the jump-pointer prefetching source and Josep Torrellas for feedback on this work.

eferences

[1] T. Alexander and G. Kedem. Distributed Prefetch-buffer/Cache Design for High Performance Memory Systems. In *Proc. of the 2nd Intl. Symp. on High-Performance Comp. Arch.*, 1996.

[2] M. Bekerman et al. Correlated Load-Address Predictors. In *Proc. of the 26th Intl. Symp. on Comp. Arch.*, 1999.

[3] D. Burger, S. Kaxiras, and J. R. Goodman. DataScalar Architectures. In *Proc. of the 24th Intl. Symp. on Comp. Arch.*, 1997.

[4] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Concious Data Placement. In *Proc. of the 8th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, 1998.

[5] M. C. Carlisle and A. Rogers. Software Caching and Computation Migration in Olden. In *Proc. of the 6th Principles and Practice of Parallel Programming*, 1995.

[6] J. Carter, W. Hsieh, L. Stoller, M. Swanson, and L. Zhang. Impulse: Building a Smarter Memory Controller. In *Proc. of the 5th Intl. Symp. on High-Performance Comp. Arch.*, 1999.

[7] T.-F. Chen. An Effective Programmable Prefetch Engine for On-Chip Caches. In *Proc. of the 28th Intl. Symp. on Microarch.*, 1995.

[8] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 1995.

[9] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Concious Structure Layout. In *Proc. of the SIGPLAN'99 Conf. on Programming Language Design and Implementation*, 1999.

[10] D. G. Elliot and W. M. Snelgrove. Computational Ram: A Memory-SIMD Hybrid and its Application to DSP. In *Proc. of the Custom Integrated Circuits Conf.*, 1992.

[11] M. Gokhale, B. Holmes, and K. Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer*, April 1995.

[12] M. Hall et al. Mapping Irregular Applications to DIVA, a PIM-based Data-Intesive Architecture. In *Proc. of Supercomputing'99*, 1999.

[13] C. J. Hughes. Prefetching Linked Data Structures in Systems with Merged DRAM-Logic. Master's thesis, University of Illinois at Urbana-Champaign, May 2000. URL: http://rsim.cs.uiuc.edu/ ~cjhughes/cjhughesmsthesis.pdf.

[14] C. J. Hughes and S. Adve. Supplemental Data for "Memory-Side Prefetching for Linked Data Structures". URL: http://rsim.cs.uiuc.edu/~cjhughes/alternative-arch.ps.

[15] IBM. URL: http://www.research.ibm.com/bluegene/comsci.html.

[16] Intel. *Intel IA-64 Architecture Software Developer's Manual*, 2000.

[17] Intel. *IA-32 Intel Architecture Software Developer's Manual*, 2001.

[18] B. L. Jacob and T. N. Mudge. A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations. In *Proc. of the 8th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, 1998.

[19] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proc. of the 24th Intl. Symp. on Comp. Arch.*, 1997.

[20] Y. Kang et al. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proc. of the 1999 Intl. Conf. on Comp. Design*, 1999.

[21] M. Karlsson, F. Dahlgren, and P. Stenström. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proc. of the 6th Intl. Symp. on High-Performance Comp. Arch.*, 2000.

[22] P. M. Kogge. The EXECUBE Approach to Massively Parallel Processing. In *Proc. of the 1994 Intl. Conf. on Parallel Proc.*, 1994.

[23] N. Kohout, S. Choi, and D. Yeung. Multi-Chain Prefetching: Exploiting Natural Memory Parallelism in Pointer-Chasing Codes. Technical Report UMD-SCA-TR-2000-01, University of Maryland at College Park, 2000.

[24] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, September 1997.

[25] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. Eighth Symp. on Comp. Arch.*, pages 81–87, May 1981.

[26] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software Prefetching in Pointer- and Call-Intesive Environments. In *Proc. of the 28th Intl. Symp. on Microarch.*, 1995.

[27] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. of the 7th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, 1996.

[28] C.-K. Luk and T. C. Mowry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation. In *Proc. of the 26th Intl. Symp. on Comp. Arch.*, 1999.

[29] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *Proc. of the 10th Intl. Conf. on Supercomputing*, 1996.

[30] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. of the 5th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, 1992.

[31] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proc. of the 25th Intl. Symp. on Comp. Arch.*, 1998.

[32] V. S. Pai, P. Ranganathan, H. Abdel-Shafi, and S. Adve. The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors. *IEEE Transactions on Computers, special issue on caches*, February 1999.

[33] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM Reference Manual version 1.0. Technical Report 9705, Dept. of Elec. and Comp. Eng., Rice University, August 1997.

[34] S. S. Pinter and A. Yoaz. Tango: a Hardware-based Data Prefetching Technique for Superscalar Processors. In *Proc. of the 29th Intl. Symp. on Microarch.*, 1996.

[35] A. Roth, A. Moshovos, and G. S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. of the 8th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, 1998.

[36] A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proc. of the 26th Intl. Symp. on Comp. Arch.*, 1999.

[37] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proc. of the 23th Intl. Symp. on Comp. Arch.*, 1996.

[38] S. Wallach. Billions and billions. 4th Intl. Symp. on High-Performance Comp. Arch., 1998. Keynote address.

[39] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Comp. Arch.*, pages 24–36, June 1995.

[40] C.-L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *Proc. of the 2000 Intl. Conf. on Supercomputing*, May 2000.

[41] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proc. of the 22th Intl. Symp. on Comp. Arch.*, 1995.